BG/L Optimization Tips

Andrew Siegel
Argonne National Laboratory

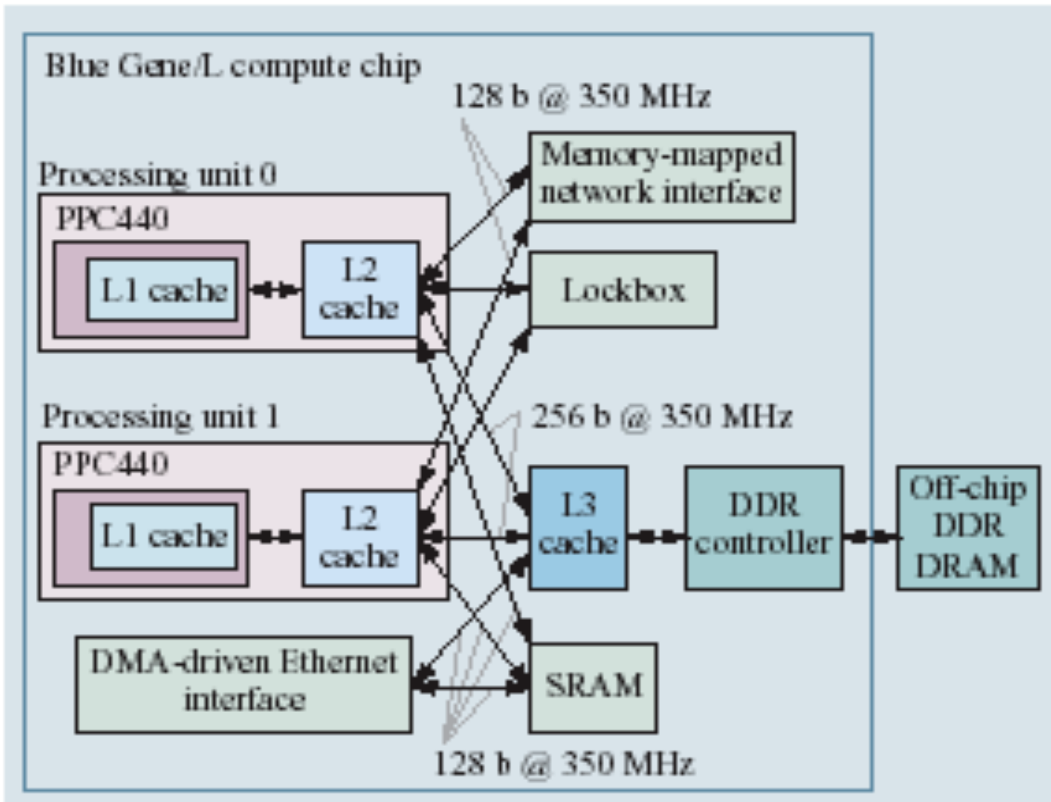
Practical Optimization Steps

- Start with those that require no code modification
  - Compiler switches
  - Virtual-node vs. Co-processor mode
  - Using optimized libraries (DGEMM, MASSV, etc.)
  - Parallel opts
    - MPI_EAGER_LIMIT
    - Explicit mapping
    - Etc
- Use directives within code
  - Alignment assertions
  - Aliasing assertions
  - Loop unrolling suggestions
  - Vectorization suggestions

Practical Optimization Steps

- Hierarchy of direct code modifications
  - appropriate if performance bottlenecks are highly concentrated
  - Rearranging memory
    - Cache reuse
    - Contiguous pairs of doubles allow for quad-word loads
  - Use double-hummer intrinsics
    - Register/instruction schedule still done by compiler
  - Hand-Coding assembler

BG/L Compute Chip



PPC440 Characteristics

- 32-bit architecture at 700 MHz
- single integer unit
- single load/store unit
- special double floating-point unit (double hummer)
- Floating-point pipeline : 5 cycles
- Floating-point load-to-use latency : 4 cycles

Double FPU

- Double FPU has 32 primary floating-point registers, 32 secondary floating-point registers, and supports :
  - standard PowerPC instructions, which execute on fpu0 (lfd, fadd, fmadd, fadds, fdiv, …), and
  - SIMD instructions for 64-bit floating-point numbers (lfpdx, fpadd, fpmadd, fpre, …)

Compute Chip Characteristics

- L1 Data cache
  - 32 KB total size, 32-Byte line size, 64-way associative, round-robin replacement
- L2 Data cache
  - prefetch buffer, holds 16 128-byte lines
- L3 Data cache
  - 4 MB, ~35 cycles latency, on-chip
- Memory :
  - 512 MB DDR at 350 MHz, ~85 cycles latency

Peak Flop/s

- 700 Hz * 2 flops/cycle * 2 fpus =
  <u>2.8 GFlop/s theoretical peak per processor</u>

- Assumes quite a few things:
  - All FMA's
  - Perfect use of double hummer (more soon)
  - Significant cache reuse (e.g. not streaming)
  - Not load bound
  - Can fill 5-stage pipeline
  - etc.

- Caution: %-peak is only meaningful in comparison to something.
  - 10% may be good, 1% may be good, 50% may be bad …

Memory bandwidth

- L1-cache: can complete 1 quadword load per clock cycle: 16B*700/s = 11.2GB/s

- Out of L1-cache: Depends on complex three-level memory hierarchy. Theoretical max = 3.7GB/s

Memory bandwidth and latency using different components

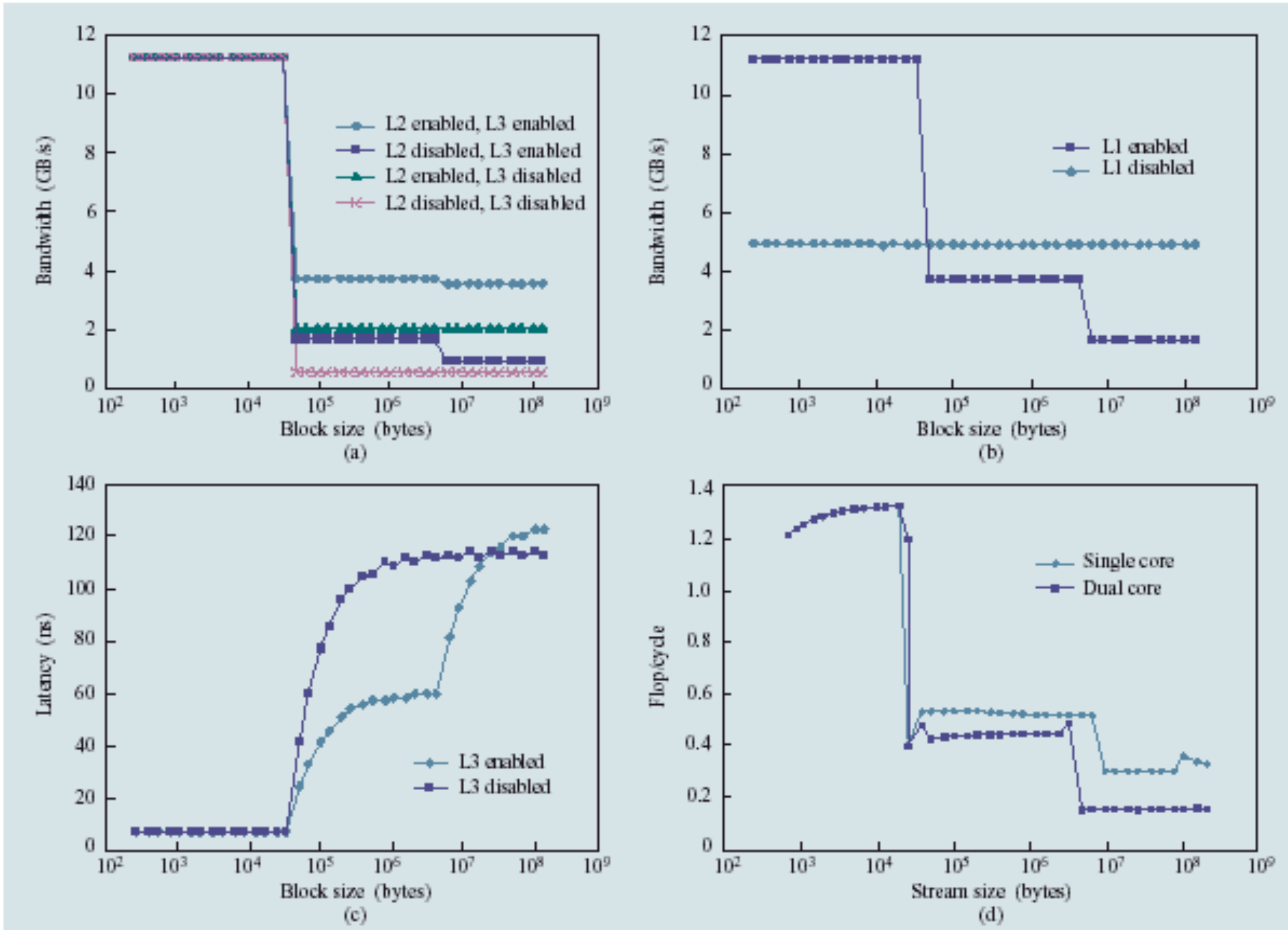


Figure 4

(a) Sequential read bandwidth. (b) Sequential write bandwidth. (c) Random access latency. (d) DAXPY performance. (a) and (b) © 2004 IEEE. Reprinted from [7] with permission.

IBM XL Compiler optimizations

- General optimization levels:
  - Default optimization = none (very slow)
  - -O : good place to start, use with -qmaxmem=64000
  - -O2: same as -O
  - -O3 -qstrict : can try more aggressive optimization, but must strictly obey program semantics
  - -O3: aggressive, allows re-association, will replace division by multiplication with the inverse
  - -qhot : turns on high-order transformation module will add vector routines, unless -qhot=novector
  - -qreport=hotlist to see vectorization report
  - -qipa : inter-procedure analysis. May cause very slow compilation.

Compiler opts, cont.

- Architecture flags:
  - -qalign=… (fortran only)
  - -qarch=440 : generates standard powerpc instructions
  - -qarch=440d : will try to generate double FPU code

- Suggested steps On BG/L
  - -O -qarch=440 -qmaxmem=64000
  - -O3 -qarch=440/440d
  - -O4 -qarch=440d -qtune=440 (or -O5 …)
  - -O4 = -O3 -qhot -qipa=level=1 -qarch=auto
  - -O5 = -O3 -qhot –qipa=level=2 -qarch=auto

- Use –v flag or check .lst file to see all flags used in compilation

Compiler Listing

- -qsource –qlist
  - Creates .lst file containing assembler listing
  - Highly recommended when trying to squeeze performance out of numerical kernel
  - Try different compiler flags and study code that is generated to understand performance

Runtime mode

- Virtual-node mode
  – Each processor on a node runs as its own MPI task and gets ½ total RAM (256MB each).
  – Use *cqsub -m vn*
- Co-processor mode
  – One CPU is used for message passing and the other for computation.
  – Compute processor gets full 512Mb RAM
  – Use *cqsub –m co*

Optimized libraries

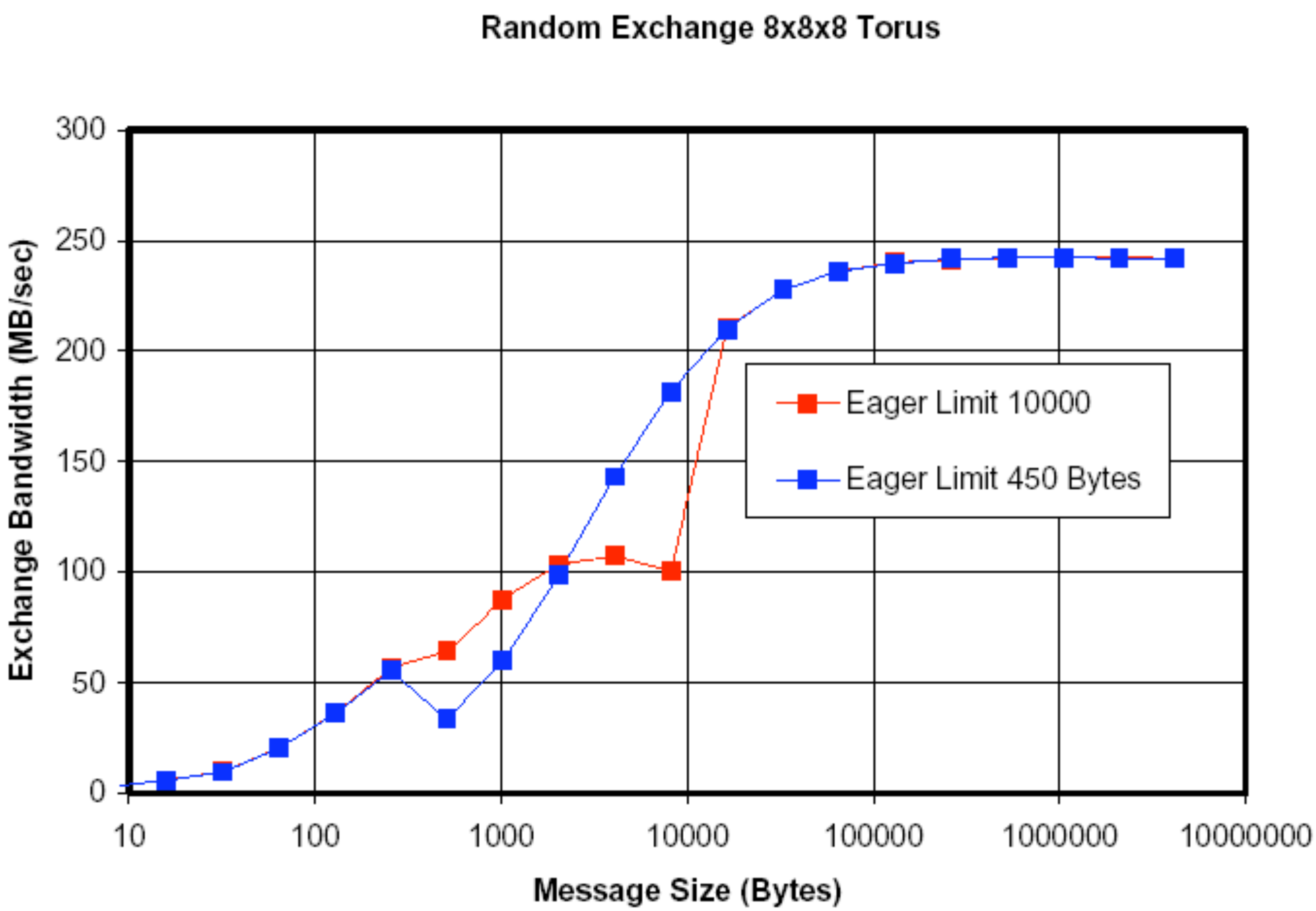
- ESSL BG/L port recently completed
  - Not much feedback yet.
- No plans for PESSL port
- Vanilla version of ESSL routines (BLAS, LAPACK, FFTW, etc.) perform poorly.
- See cheatsheet for full details/examples

MPI Mapping

- With virtual node mode, experiment with
  - BGLMPI_MAPPING=TXYZ
  - This puts tasks 0 and 1 on the first node, tasks 2 and 3 on the next node, with nodes in x,y,z torus order.
  - The default layout is XYZT, which is often less efficient than TXYZ.
  - Also note that in TXYZ mode, you get two tasks per node if you have #tasks < 2*#nodes; otherwise the XYZT layout will leave just one task on at least some nodes.
  - Can also write a mapfile to explicitly control processor mapping

EAGER_LIMIT

- BG/L can route messages either statically or dynamically
- By default small messages (those smaller than MPI_EAGER) are routed statically, and large ones are routed dynamically
- These can be controlled with the following environment variables (see cheatsheet)
  - BGLMPI_EAGER = 1000 (default is 10000)
    - Sets limiting message size in bytes for eager protocol
  - BGLMPI_AE = 1
    - To try adaptive route for eager message. Default is static.


Random Exchange 8x8x8 Torus
Exchange Bandwidth (MB/sec)
300
250
200
150
100
50
0
10
100
1000
10000
100000
1000000
10000000
Message Size (Bytes)
Eager Limit 10000
Eager Limit 450 Bytes

Compiler assertions

- Three compiler assertions are particularly important for generating optimal code
  - Alignment
    - call alignx(16,x(1)) Fortran
    - __alignx(16,x)   C
      - Inform compiler that variable x is aligned on a 16-byte boundary.
  - Aliasing
    - #pragma disjoint(*a,*b) C only
      - Inform compiler that a and b will not refer to overlapping memory
  - Unrolling
    - !ibm* unroll(n) Fortran
    - #pragma unroll(n) C
      - Unroll inner loop that follows n elements

Example with DAXPY

- Fortran

```
call alignx(16,x(1))
call alignx(16,y(1))
!ibm* unroll(10)
do i = 1, n
  y(i) = a*x(i) + y(i)
end do
```

- C

```
double * x, * y;
#pragma disjoint (*x, *y)
__alignx(16,x);
__alignx(16,y);
#pragma unroll(10)
for (i=0; i<n; i++) y[i] = a*x[i] + y[i];
```

Double-hummer examples

- See ~siegela/examples/
  - mxm
    - In-cache matrix-matrix products using double-hummer intrinsics
  - dotp
    - dot product using double-hummer intrinsics and ensuring alignment
  - ax+b

Listing file

- Use –qsource –qlist to generate friendly assembler listing
- Good strategy is to tweak source, compiler options and diagnose with .lst output, rather than hand-coding assembler.

Performance Tools

- Currently installed performance tools
  - gprof for per-routine timings
  - memmon for detecting high-water memory mark
  - mpitrace for automatically timing mpi calls
  - hpmlib preliminary port
  - papi preliminary port
  - tau for more integrated and complex analysis
    - Requires PAPI or hpmlib for hardware counters
- See cheatsheet for examples of how to use